Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Matias Kuusela

Architectural Design Decisions in Agile Software Development Teams

Master's Thesis
Espoo, May 27, 2015

Supervisor: Professor Marjo Kauppinen
Advisor: Mika Kivilompolo Ph.D. Floweb Oy
Varvana Myllärniemi M. Sc. Aalto University

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Matias Kuusela

Title:
Architectural Design Decisions in
Agile Software Development Teams

Date: May 27, 2015 **Pages:** 59

Major: Software Engineering and Business **Code:** T-76

Supervisor: Professor Marjo Kauppinen

Advisor: Mika Kivilompolo Ph.D.
Varvana Myllärniemi M.Sc.

Architectural design decisions have been a focal point of architectural research for years. Yet our understanding of how such decisions are made in agile development teams is limited to assuming that tightly knit developers make such decisions together. Even this assumption goes out the window when the development team is not co-located. There is also little understanding of when architectural decisions are made with different methods giving contradicting recommendations.

The goal of this thesis is to show how architectural design decisions are made in one distributed agile development team. The research was done using action research methodologies due to their strength in understanding groups that are not very strictly structured.

In the studied team independent developers have the initial responsibility for architectural design decisions but can share this responsibility with other members of the development team if necessary. We will also show the product owners role in supporting the developers in decision making and how the decisions that have been made are communicated to the product owner and the development team. In addition we present three project phases for architectural decision making: planning, foundation and iterative.

We will show how problems arise when the developers fail to identify architectural design decisions or are unable to get the help they need for making them. We will also show how diverting understanding on how the system should work between the product owner and development team can lead to great problems. Based on these results we introduce a set of communication and development recommendations for alleviating these problems.

We conclude that although empowering individual developers in making architectural design decisions carries many benefits it also puts heavy stock on both the abilities of individual developers in identifying and making architectural design decisions, but also in the team's ability to work together making of these decisions. Distributed teams should especially take these challenges into account in their development process.

Keywords: software architecture, architectural design decissions, agile software development, Scrum

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Matias Kuusela

Työn nimi:
Arkkitehtuuriset suunnittelupäätökset ketterissä ohjelmistokehitys ryhmissä

Päiväys:	27. Toukokuuta 2015	**Sivumäärä:**	59
Pääaine:	Ohjelmistotuotanto ja liiketoiminta	**Koodi:**	T-76
Valvoja:	Professori Marjo Kauppinen		
Ohjaaja:	Tohtori Mika Kivilompolo Diplomi-Insinööri Varvana Myllärniemi		

Arkkitehtuuriset suunnittelupäätökset ovat olleet keskeinen osa ohjelmistoarkkitehtuurin tutkimusta jo vuosikymmenen ajan. Kuitenkin ymmärryksemme siitä miten arkkitehtuuriset suunnittelupäätökset tehdään ketterissä kehitysryhmissä rajoittuu olettamukseen, että tiiviisti toimivat ketterät kehitysryhmät tekevät nämä päätökset yhdessä. Edes tämä oletus ei päde hajautettujen ryhmien tapauksessa.

Tämän diplomityön tavoitteena on kuvata miten arkkitehtuuriset suunnittelupäätökset tehdään yhdessä ketterässä kehitysryhmässä. Tutkimus toteutettiin toimintatutkimuksena sillä se sopii hyvin rakenteellisesti löyhien ryhmien tutkimiseen.

Tutkitussa kehitysryhmässä vastuu arkkitehtuurisista suunnittelupäätöksistä on yksittäisillä kehittäjillä. Kuitenkin kehittäjät voivat tarvittaessa jakaa tätä vastuuta muille kehittäjille. Osoitamme myös kuinka tuoteomistajalla on suuri rooli päätösten teon tukena ja näytämme kuinka tehdyt päätökset kommunikoidaan muille kehittäjille ja tuoteomistajalle. Lisäksi kuvailemme arkkitehtuuristen päätösten tekemistä projektin kolmessa eri vaiheessa, jotka ovat suunnitteleva, perustava ja iteratiivinen.

Arkkitehtuuriin vaikuttavat suunnittelupäätökset nousevat ongelmaksi, kun kehittäjät eivät tunnista suunnittelupäätöksiä arkkitehtuurisesti merkittäväksi tai kun kehittäjät eivät saa tarvitsemaansa tukea päätösten tekemiseen. Työssä osoitetaan myös kuinka tuoteomistajan ja kehittäjien yhteisen näkemyksen puute kehitettävästä tuotteesta johtaa suuriin ongelmiin arkkitehtuuristen päätösten tekemiseksi. Näiden ongelmien lievittämiseksi esitetään joukko parannusehdotuksia.

Johtopäätöksenä todetaan, että vaikka on edullista antaa yksittäisten kehittäjien tehdä arkkitehtuurisia päätöksiä, tämä vaatii kehittäjiltä kykyä tunnistaa ja tehdä arkkitehtuurisia suunnittelupäätöksiä sekä kehitysryhmältä mahdollisuuksia toimia yhdessä. Varsinkin hajautettujen kehitysryhmien tulisi ottaa nämä seikat huomioon.

Asiasanat: ohjelmistoarkkitehtuuri, arkkitehtuuriset suunnittelupäätökset, ketterä ohjelmistokehitys, Scrum

Kieli: Englanti

Acknowledgements

I would like to thank my supervisor Marjo Kauppinen and advisor Varvana Myllärniemi for driving this work towards academic excellence. I would also like to express my gratitude for Mika Kivilompolo for arranging the opportunity for this case and supporting the research work at Floweb Oy. I would also like to express my gratitude for all members of Floweb Oy who spent their time in the interviews that formed the backbone of this work. Finally I would like to thank my family for all of their support.

Espoo, May 27, 2015

Matias Kuusela

Contents

Chapter 1

Introduction

1.1 Background

Software architecture is concerned with structure, organization and interactions of the components of modern software systems. Although the origins of the term software architecture are in the 1960's, software architecture emerged into its own discipline in the early 1990's. [24].

Software architecture did implicitly record the rationale behind architectural decisions from the start. However this information tends to evaporate over time, leading to architectural erosion as the architecture evolves, as the previous concerns are not taken into account. The idea of explicitly recording and studying architectural design decisions was brought up by Jansen and Bosch [19] in 2005 to help prevent architectural knowledge loss and erosion.

A group of lightweight software development methods appeared in the 1990's as a counterweight to waterfall software development method. The proponents of these methods would join together in 2001 to create an agile manifesto, after which these methods have been known as agile methods. These methods share iterative development as their core, emphasizing lightweight planning and interaction between developers as well as between customers and developers.

1.2 Motivation

Architectural design decisions have been one of the topics of interest in software architecture for a decade now. Yet our understanding of how such decisions are made in agile organizations is limited. We either assume that the developers and customers identify relevant needs of the system and then come up with an architecture that suits those needs [28] or that an architect

of species Architectus Oryzus oversees the developers and guides them to making great architectural design decisions [15].

Situation is made even more complex when the team is not co-located and therefore cannot always make the decisions together with the customer representative or under the architects guidance. Since distributed software development has become common even in agile development [31] we would benefit from more detailed understanding on how distributed teams make architectural design decisions.

1.3 Research problem

The goal of this work is to describe how architectural design decisions are made in a small distributed agile team. We will answer the following research questions:

1. How are architectural design decisions made?
2. How are the architectural design decisions that have been made communicated?
3. What problems arise from the way architectural design decisions are made and communicated?
4. How could these problems be solved?

These questions are answered based on action research conducted in Floweb OY for which the author has worked for two years. As is typical of young agile organizations, Floweb Oy does not strictly follow any specific agile method, having picked up practices from various methods. This ill-defined nature of the organization makes it ideal ground for action based research.

1.4 Structure of the Thesis

This thesis is divided into six chapters. Chapter 1 introduces the research question as well as the motivation and background. Chapter 2 describes the organization that is being studied, methodology used to carry out the research as well as what was done during the research. Chapter 3 reviews prior scientific research and literacy in the fields of architectural design decisions, agile and agile architecture. Chapter 4 presents a brief look into the current

state of the organization the research is carried in as well as findings pertinent to the research questions. Chapter 5 answers the research questions based on the results of the research and reflects these results with literacy. Chapter 6 gives the conclusions of this research and presents some future topics of interest.

Chapter 2

Research method

2.1 Case description

Floweb Oy (company) is a software development and service operator. Founded in 2012 the company is jointly owned by three individual founders and a major Finnish flower importer (parent company). Currently the company operates the web shop of the parent company which sells products to individual flower shops. The future goal of the company is to expand to operating business to consumer shops. This goal is to be achieved by developing and operating a consumer web shop system that partner flower shops can then use to sell their products directly to consumers. The start of this development process coincided with the start of the research for this thesis.

The company employs three developers, including the author (DEVS 1-3 in 2.1). Two of the developers including the author work full time, albeit both are doing their master thesis. The third developer works from half to fulltime basis. Additionally to the developers the three founders of the company take care of the product owner, scrum master and architect roles

Code	Role	Employment	Note
Dev 1	Developer	full-time	
Dev 2	Developer	full-time	author
Dev 3	Developer	part-time	
SM	Scrum Master	irregularly	
PO	Product Owner	irregularly	
ARCH	Architect	irregularly	

Table 2.1: Development team

in the development organization (SM, PO and ARCH in 2.1). Due to the financial limitations of the company the founders can only work in these positions on the side of their day jobs and thus their contributions vary greatly.

The developers have an office space allocated to them which they share with personnel from the mother company. Commonly the DEVS 1 and 2 will work together on couple of days a week, while the DEV 3 tends to come to the office once every two weeks. The SM and ARCH tend to visit the office about every second week as well with the architect usually working with the team for the full day when he does. The PO is located over 500 kilometers away from rest of the team and tends to visit very irregularly, usually a couple of times a year.

2.2 Action research approach

2.2.1 Method overview

The purpose of this research is to find out how architectural design decisions are made in a distributed agile organization and how they could be improved. The organization in question uses agile in a way that is in the author's opinion typical of smaller organizations: various methods from various agile methodologies are adapted and then fall out of use or are only used by some members of the team. Since action research is strong in looking at what organizations are actually doing rather than what they say they are doing [1], the thesis writer considers it the most valid methodology for carrying out research in this context.

Action research also has the advantage of offering results with high relevance [3], while information system research tends to suffer from poor relevance [33]. In the past action research has been criticized for its lack of methodological rigour [11], for the fact that action research can be hard to distinguish from consulting [1] and the tendency that action research produces either "research with little action or action with little research" [13]. This work attempts to avoid these problems by following the cyclical action research methodology as well taking into account the action research principles presented by Robert Davison et. al. [12].

The action research cycle used in this project is the Susman and Evereds [37] action research cycle, one the best known action research methods [18]. The phases used in this work are:

Diagnosing Understanding the current state of the organization, the problems the organization faces and their causes

action planning Finding a course of action for solving a problem in organization

action taking Introducing a course of action to the team and observing its effects

evaluating Evaluating the effects of the intervention

reflection Reflecting on the research process

In optimal case every cycle would contain all of the phases so that a cycle ends in reflection and new one starts from diagnosis, however in practice taking shortcuts is often necessary [12]. In the next section we will present how these phases were carried out in this work.

Phase	Action taken with team	Research
diagnosis I	team interviews	literacy review
action planning I	discussing alternative actions with team	analyzing current situation
action taking I	introducing first set of improvements	initial results for research questions
evaluating I	evaluating the effectiveness of first intervention phase	results for research questions
action planning II	setting up for second intervention	initial answers to research questions
action taking II	introducing second intervention	reflecting research question answers with literacy
evaluating II	evaluating the effectiveness of second intervention	updating results
reflection		additional literacy review, updating answers and conclusions

Table 2.2: Research phases

2.2.2 Research process

The research carried out for this thesis followed the outlines of canonical action research presented before while taking into account the requirements of master thesis work (need for both literacy and empiric research).The research done is presented in table 2.2 with phase marked on first row, a summary of action taken with the team in second row and research done on third row.

The research process started with a diagnosis phase. This consisted of background research into architecture, agile methods and agile architectures, results of which formed the backbone for the literacy chapter of this thesis.At the same time author interviewed the members of the development team (table 2.4 to learn of the current state of the company and how members of the team aproached architectural design decissions. The interview questions are in appendix A, original finnish questions are provided since all but one interview was conducted in finnish.

The intervention consisted of two phases (rows 2-7 in table 2.2) . Based on the interviews and additional discussions with the team the author and team agreed on three solutions (first three rows in table 2.3). After these were tried for about a month the author evaluated the effectiveness of these solutions. Since the solutions were not having as much effect as had been hoped the author introduced one more solution (last row in table 2.3) that based on the

Solution	Round introduced in
A:S1 Weekly meetings between developers and architect	I
B:S1 Monthly face to face meetings between developers and the PO	I
B:S2 Continious deployment of new code	I
A:S2 More co-located development	II

Table 2.3: Solutions

Interviewee	Role	Recorded	Transcribed	Length (minutes)
Dev 1	Developer	YES	YES	39
Dev 3	Developer	YES	YES	24
SM	Scrum Master	YES	YES	56
PO	Product Owner	YES	YES	43
Arch	Architect	YES	YES	75

Table 2.4: Interviews

interviews and observations made during the first phase. Another round of intervention was committed after which the full intervention was evaluated. As can be seen from table 2.2 there was no time to run a full cycle between phases, since there were only couple of days between interventions.

During the interventions the author answered the research questions based on the interviews, additional discussions with team and observations made. Finally once the intervention was over there was need for some additional literacy review to properly reflect the results with the literacy. Finally conclusions were drawn from the results of the research and the author reflected the effectiveness of the research.

Chapter 3

Literacy review

3.1 Architectural design decision

3.1.1 Software architecture

Despite its key role in software engineering [14] the research community has had difficult time in finding a single commonly agreed definition for software architecture. During the recent years this has evolved into two distinct ways of approaching software architecture [30]: a structural view and an architectural design decision view.

The structural view it self has during the years had many different definitions. The most modern of these is the ISO/IEC/IEEE standard 42010 [17] for architecture as: " fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution". The main evolution of the ISO 42010s definition of software architecture compared to earlier definitions, such as that from L Bass [9], is that it is explicitly concerned with the communication, design and evolution of the architecture.

The second approach is to think of architecture as a set of architectural design decisions and the rationale behind these decisions [19]. This approach has the advantage of being fairly light weight compared to the structural approaches. It is also very suitable for looking into agile architectures since it does not require us to consider architecture as a documented abstraction but rather as a stream of decisions [29]

It is important to note that architectural design decisions are a sub set of design decisions that are architecturally significant [21]. To understand the concept of architectural design decisions we should consider when decisions are architecturally relevant. There are various different ways to approach this.

Jansen defines design decisions to be architecturally significant when they directly effect the design of the software architecture [21]. Going back to the ISO 42010 definition of architecture we can conclude that any decision that effects the fundamental organization of components of a system or the relations between the components is architecturally significant.

This aproach has the weakness in that it is very difficult to determine what exactly is a component in a system that has various structures all the way to the code level. How do we define which of these structures are the components that form the fundamental organization of the system? One way to approach this is to consider that things that are difficult to change once implemented are architecturally significant, since they are the things that you want to get right before their implementation [15]. Thus we consider those elements that are hard to change to be the fundamental components of the system.

Another interesting approach to the components on different layers of abstraction is presented by Malan and Bredemeyer [26] advocating that we should embrace this layering. In their model architecture is divided on component, application, domain and enterprise scopes and architecturally significant design decision are architecturally significant only if they cannot be deferred to a lower level. Design decisions that can be handled on component level are no longer architecturally significant but can be taken care of by the designer or developer responsible for that component. This way the architect does not needlessly restrict the designers and developers ability to do their work.

Yet another approach is to consider architectural significance as a factor of cost and risk associated with the decision [30]. In this model the architects primary concern is to manage the cost and risk factors of architectural design decisions. Thus the economic factor of the decisions should be the architects primary concern [30].

3.1.2 Elements of architectural design decision

We have already considered what makes a design decision an architectural design decision. But what do architectural design decisions consist of? Various models have been proposed over the years. Shahin et al. [35] have identified three elements that all architectural design decision models have in common: constraint, solution and rationale. Most models also include some form of motivation for the design decision, and it makes the fourth item in our consideration.

1. Constraint: Refers to the factors that influence the design decisions,

for example the various quality attributes and the limits of the system itself.

2. Solution: Can be seen to equal the decision itself. It is the active part of the design decision, effecting the system and its architecture.

3. Rationale: The reasoning for the decision. Why was the decision chosen over the alternatives?

4. Motivation: The impulse for making the decision. The decision can for example be seen to solve a problem or fulfill a requirement.

In the next section we will see that how motivation, constraints and rationale drive the architectural design decisions. In 3.1.4 we see how the chosen solution is described.

3.1.3 Architecting

As there are multiple different ways to approach architecture so there are multiple different ways to approach architecting. ISO 42010 [17] defines architecting as "process of conceiving, defining, expressing, documenting, communicating, certifying proper implementation of, maintaining and improving an architecture throughout a systems life cycle (i.e., "designing")".

Essentially these boil down to two different aspect - the designing of software architecture during the different phases of the project (conceiving, defining, certifying, maintaining and improving) and communication of the architecture via different mediums (expressing, documenting, communicating). It is the approach to architectural design as a phased process and focus on abstracted document based communication that sets this approach apart from architectural design decision approach.

Jansen et al. [20] define the architectural design process as an iterative process with a problem and solution scopes. The problem scope includes the system that is being designed as well as the various issues, requirements and concerns related to the system and its development. The solution scope includes the process of making an architectural design decision.

This model has four steps:

1. Observe problem scop:. The architect observes the problem scope and comes up with an architectural problem.

2. Propose solutions: The problem motivates the architect to come up with solutions that solve the problem.

3. Choose solution: The architect chooses the best solution for the problem.

4. Modify and describe architecture: The architect modifies the systems architecture and describes the changes.

Since the last step changes the problem scope (e.g. the system) the Jansens model is essentially iterative in nature, since these changes will motivate and effect future changes.

Another more agile oriented design model is presented by Poort [29] as continuation of the cost and risk driven architecture. This model is based on a backlog of architectural concerns which are prioritized by their cost and risk. The architects responsibility is then to make those architectural design decisions that he finds to have high cost-risk factor if they are left undone. During this process the architect needs to look out for new concerns based both on the decisions that have been made but also on the changing requirements of the system. The architect also has to constantly prioritize the backlog of architectural concerns.

3.1.4 Describing software architectures

There are various different methods for describing software architecture with various different goals. Many methods record architectural design decisions directly with the goal of sharing architectural knowledge. Since we have already discussed architectural design decisions at length there is little need to describe these methods further - suffice to say that what is recorded correlates fairly well with our description in previous chapter.

The more popular way to describe software architecture is to describe the system, its components and their interaction in as an abstraction - typically using visual means. Most commonly these methods use what are called architectural views, introduced by Philippe Kruchten [25]. The purpose of architectural views is to offer different stakeholders a view into the system by using various levels of abstraction.

Although Kruchten's 4+1 model is fairly popular, there is no firm consensus on what views should be used. Indeed many practitioners advocate that views should be chosen per project [8]. However for future reference some of the more popular view models have been listed below.

Krutchen 4+1

1. Logical view: Describes the system as objects, often matching the classes of object oriented programming classes. Various levels of ab-

straction can be used, with multiple objects, or classes, grouped into object groups.

2. Process view: Describes the running of various processes that make up the system. Usually concerned with non-functional requirements such as systems performance and availability.

3. Development view: Used to divide the system into subsystems. Mostly concerned with the development practices used to develop these systems. Also used to allocate work between different developers.

4. Physical view: Mapping the software into the physical world, commonly to the various servers and clients running the software. These days more commonly known as deployment view.

5. Scenarios: The architectural descriptions are illustrated using a set of Scenarios - a selected group of vital use cases. The purpose of these is to both validate the existing architecture as well as to bring up new architectural concerns.

Siemens model [16]

1. Conceptual view: Describes the major design elements of the system and their relations. Concerned with making sure that software matches its requirements and all of the components can be integrated into one system.

2. Execution view: Describes the distribution and running of the software. Mapping of conceptual module components to run-time entities.

3. Code view: The organization of the source code: the various libraries and source objects, as well as binary and source files and their directory structure.

4. Module view: Division of software into modules and of the modules into layers. Used for handling the complexity of the software and division of labor.

It is easy to see that these models have many common elements. Indeed in the next chapter we will see that the various views of software architecture are easy to map to the use cases of software architecture.

3.1.5 What is software architecture used for

Software architecture can be used for various things, and what it is actually used for depends heavily on the development organization [2]. In his doctoral thesis work on architectural design decisions Anton Jansen [20] presented five ways to use software architecture:

1. Blue-print: Outline of the system being designed, i.e. a blueprint. With additional detailing this can be used as a basis for the actual implementation of the system. Arguably the primary use case for software architecture.

2. Roadmap: A way to predict and make design decisions that are not immediately relevant, e.g. to plan ahead. This allows the software architecture to better match the company?s long term business strategy.

3. Communication vehicle: Allows the sharing of architectural design decisions between different members of the development team, thus allowing them to contribute to the decision.

4. Work divider: As software architecture decomposes the system into smaller parts, it can be used to divide the development of the system to different developers.

5. Quality predictor: Software architecture can be used to predict how well the system will match the quality attributes demanded from the system. This allows the system to be designed to match the demands placed on it.

In addition we will consider a sixth usage for software architecture.

1. Risk management: Software architecture can be used to predict possible problems faced by the system either during its development or during its use. Architectural risk management not only includes risk avoidance, but also the prediction of the severity and likelihood of various risks.

As we mentioned in previous chapter, these use cases are clearly presented in the various view models. The main concern with both Siemens and 4+1 model is creating a Blueprint for the system. In addition all of the views are used in communication with the development team.

Both view models have a view that are used for dividing work, Development view in 4+1 and module view in Siemens model. The conceptual view

in Siemens model is used as a communication channel with various stakeholders, while the scenarios in 4+1 are used to gather requirements from various stakeholders. The conceptual view also acts as a Roadmap for the Siemens model with its focus on requirements, while the Scenarios do the same for 4+1 model. Krutchens process view is focused on managing non-functional quality requirements, and while not explicitly stated it can be assumed that execution view does the same for Siemens model.

3.2 Agile software development

3.2.1 Agile movement

The agile movement is a very diverse movement encompassing different software development methods and ideas. Indeed there are often arguments on whatever a software development method is agile or something else entirely. The most critical of these borderline cases being lean software development (B. Veli. 2010.). For clarity's sake this work considers Lean and other agile like methods agile.

What unifies agile movement are the principles of the Agile Manifesto [6]. In the manifesto the agilists renounced software development methods based on strict processes and demanded that software development should focus on the development of functional software rather than on documentation for documentations sake. The agilists also emphasized that good software arrives from the development teams strengths and inter team communication. In its own words the manifesto [6] states:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

Another unifying theme for the agile methods is iterative software development. Rather than having distinct phases for different parts of software development (f. ex. planning, coding and testing) iterative methods have repeating iterations which contain all software development activities. The

different methodologies do however differ on how these iterations are organised and how long they should be. Scrum for example recommends 4 week iterations while XP recommends iterations of two weeks [10].

Although unified by the principles of the agile declaration and iterative software development, the agile methods are very different from one another. For example Extreme programming focuses heavily on software development methods such as pair programming and integrated testing, and can be seen as more of a software development methodology rather than management methodology [10]. Scrum is concerned with organizational structure and management of development tasks and can be seen to be more interested in management of software development, which indeed is the stated goal of Schwabers [34] book "Agile Project Management with Scrum. Due to this wide diversity it is common for development teams to adapt only some parts of an agile methodologies [10] or even to combine agile methods and more traditional models such as CMM [32].

3.2.2 Architecture in agile software development

There are three major points of contention for traditional software architecture and agile software architecture:

1. The amount of planning that should be done before implementation of the software.

2. The amount of documentation that should be done for the software architecture and

3. The role of the software architect in project.

Agilists problem with doing architectural planning originates from agile principle: "Responding to change over following a plan". Many agilists initially saw software architecture as a Big Design Up Front method that lead to needless documentation [23]. Instead the agilists advocated that software architecture should emerge gradually during the iterative development process [23]. It should be noted that agile methods have never had a firm consensus over exactly how much planning should be done in advance, with XP advocating absolutely minimum amount of planning [4] and methods like Crystal advocating for a fair amount of planning up front [7].

With modern architecture more focused on architectural design decisions rather than on creating comprehensive system designs the planning conflict has simmered down during recent times, with most agilists agreeing that atleast some of the architectural design decisions should be made before

implementation. It is still difficult to say which architectural design decisions should be made in advance, with some agilists advocating that decisions with high risk factors should be focused on [30].

The iterative way of architectural design does add some requirements for software architecture. Since the architecture is not planned it should be kept as simple as possible, so as to make future changes possible [4]. On architectural design decision level this means that architectural design decisions should avoid placing constraints on future architectural design decisions. On component level this means that we should have as few architecturally significant components as possible, ergo we should try to avoid components that are difficult to change once implemented [15].

The second point of contention relates to the agile principle of "Working software over comprehensive documentation" and the sixth agile principle: "the most efficient and effective method of conveying information to and within a development team is face-to-face conversation". As we saw in chapter 2.X, traditional architectural tools such as view models were used to create comprehensive descriptions of systems architecture. This sits poorly with agilists, who prefer to rely on tacit knowledge shared by the developers rather than explicit knowledge recorded in documents. However this does not mean that agilists do not use some architectural descriptions, but these tend to be single artifacts rather than comprehensive documents [36].

There are some problems and pitfalls associated with lack of comprehensive architectural description. Most of these relate to knowledge loss that happens naturally over time as developers do not have a perfect memory or when developers leave the project. This can also be a big problem when the team starts a new project and does not take the architectural knowledge gained in the previous project into account. [36].

Finally the third point of contention for architecture in agile software development is the role of the architect himself. Agile methods tend to favor very flat organizational hierarchy with no strictly assigned roles. This arises from the agile principle "The best architectures, requirements, and designs emerge from self-organizing teams" [6].

Due to these agile principles it is usually assumed that the software architect should work with the development team and take coding tasks just like any other developer. However at the same time the architect should use his expertise to spot potential architectural problems and make sure that the system architecture stays sound.

Some Agile methods also include the possibility of not having a specific architect nominated, but rather dividing the architects responsibilities amongst the developers [4]. This can be augmented by having the developers meet up in an architectural group and discussing the various architectural issues

together.

There are two potential problems with more freeform division of architectural responsibility. First is the obvious problem that the freeform architectural division actually means that no one takes responsibility for architecture. Second is the problem of architectural knowledge splintering, since there is no single go to guy that knows what the architecture is and where it is defined.

Chapter 4

Results

4.1 Current situation

The way architectural design decisions is made is greatly affected by the organization that makes them. To facilitate this understanding we present here the organization and the software development process it uses.

4.1.1 Development organisation

Like most agile development groups, Floweb Oy has a fairly flat hierarchy. The core development team consists of three founders who work on the company on their spare time from their day jobs and three developers who work for the company either part time or full time. In addition there are contract people who work for the company when their expertise is required. This thesis focuses heavily on the core development team and its practices and thus we mostly exclude the contract workers from this work.

The role of the product owner is taken by the chief executive of the company. The Product Owner has studied floristics up to applied sciences level and has worked as an entrepreneur on the field for about seven years. In addition the product owner has taught the subject for a long time. This makes the Product Owner the far most experienced in matter pertaining to the floristics as well as consumer sales. The Product Owner role is fairly clearly set in the development organization. However in addition to taking care of the Product Owner role the Product Owner also does heavy amounts of graphical and usability work for the company as well as taking care of content creation.

The role of Scrum master is managed by a veteran information system administrator with experience from multiple information system projects. Since the development organization does not actively follow scrum practices

in its day to day development work the role of the scrum master has migrated towards a more managerial role, with heavy responsibilities on managing the financial aspects of the company. However the scrum manager still does his best to facilitate the development work by making sure that the developers get paid on time. The scrum manager also does his best to introduce some rigor into the software development process.

The Architect role is occupied by a senior software developer with over a decade of software development experience. Described as a coding architect on his day job, the architects role in the organization falls into the agile architect niche. Although the architect emphasizes that most of his time is spent on coding, he is also responsible for typical agile architectural duties such as technology choices, quality control and making sure that at least the minimal level of documentation is done, although the architect admits that the last part has not received nearly enough attention.

4.1.2 Software development process

As was described in 2.1, Floweb Oy does not use a particular holistic software development process. However a Kanban board has been used as core of the development process, with development stories flowing through the board in a free flow manner. Although efforts have been made to enforce the various Kanban rules the author has observed that the team still tends to move items backward against the Kanban item progression rules and tasks are occasionally edited on the fly when they should be closed and new ones created to replace them.

The development team does not use iterations of fixed length, although the Kanban board is occasionally trimmed. There have been attempts to fix the time gap between trimming sessions, however the team found it difficult to keep to these timetables. In fact as of late the Kanban use has dropped dramatically. This has probably been because the PO has not been taking an active role with Kanban management as before due to other responsibilities as mentioned in previous chapter.

The task creation process is fairly fluid. Most of the tasks come to the development team from the product owner. The PO has three main sources for the tasks: His own intuition in trying to turn his service idea into reality, the requests from the company's main customer pertaining to the business to business service, and finally his interactions with the end customers. The developers also create refactoring tasks when they realize that the system needs to be improved in some way. Also the Scrum Master and the Architect act as product owners of some specific systems and are responsible for task creation for them.

Tasks come from the Product Owner to the team in many different formats. The Product Owner will sometimes flesh out the task as a graphical design that the developer is to implement. On the other hand many tasks only come in as a text description of a problem that should be solved or some sort of a story on what the system should do. In these cases the Product Owner trusts the developer to follow the previous designs in creating the user interface.

In the end the technical implementation of the task is always up to the developer. The developer can discuss how best to the implement the task with other members of the development team - however it is up to the individual developer to decide if he want to do so. The management has very high level of faith in the ability of the first and second developer in getting the implementation right and thus they feel comfortable in leaving the decisions in their hands. Such a faith does not quite exist with the third developer and thus the architect is supposed to supervise his decisions more carefully.

4.1.3 Overview of company service architecture

Although this work focuses on how software architecture is created rather than on architecture itself, understanding the basic structure of company‘s service architecture makes it easier to understand how architectural decisions are made. The company‘s service architecture as presented in figure 4.3 is a replicate of one that was created and presented by the writer in the first meeting of the action phase. The services are represented as squares or circles and various links and API‘s as lines between services. Dotted lines are used in cases where the service interface has not been fully planned as of yet.

As can be seen in the figure 4.1the company has a fairly large set of different services. The two main services are the Business to Business web shop and end customer web shop - which are not directly connected with one another. The various secondary services mainly support these two services by either providing them with data or by directing customers to the services. Most of the secondary services are also accessible by end customers and thus bring value to the customers in on them self, for example the main function of image gallery is to provide product images for both web shops, but it can also be used as a standalone image gallery on its own.

We will describe some of the systems in more detail in the further chapters.

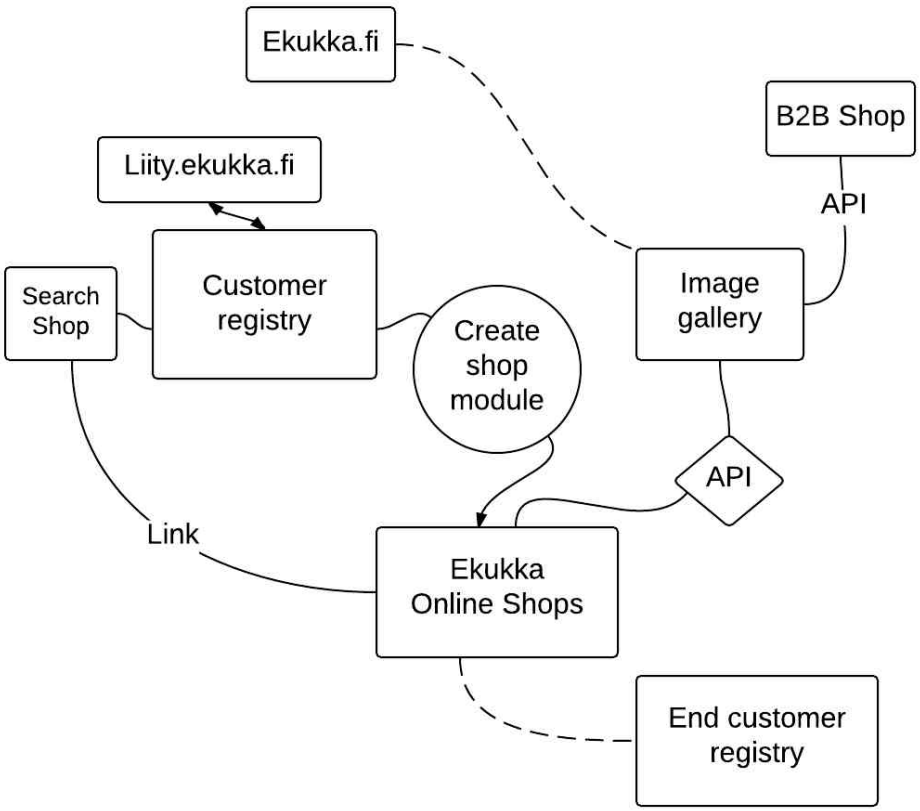

Figure 4.1: Service architecture of the company.

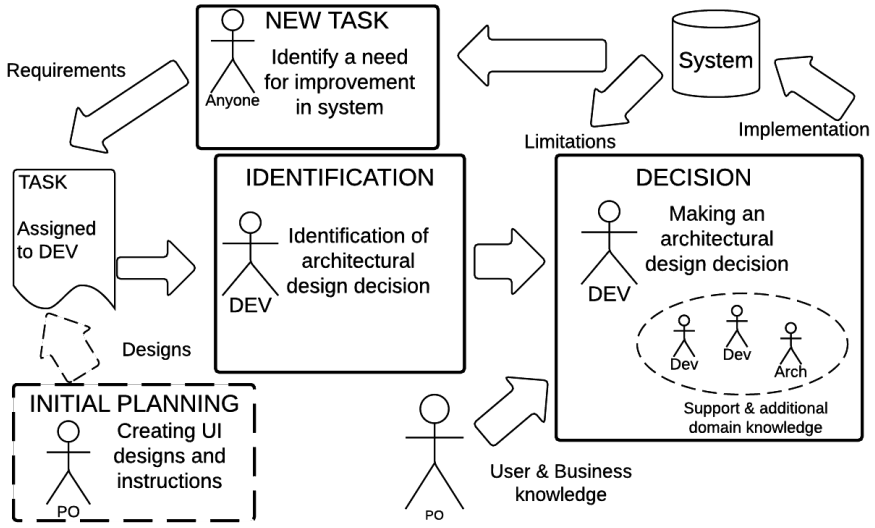

Figure 4.2: Overview of how architectural design decisions are made.

4.2 Making architectural design decisions

The process of making architectural design decisions is illusterated in figure 4.2 4.2. This process involves two major steps: identifying the architectural design decision and making an architectural design decision (center lane in figure 4.2). The responsibility for these steps lies with the developer responsible for the implementation of the architectural design decision. In this chapter we will describe these steps as well as the role played by the various product owner (PO in buttom of figure 4.2) and rest of the development team (inside circle in decision box in figure 4.2) in supporting the decision.

Additionally we will present how architectural design decisions made both limit the future architectural design decisions and create a demand for future architectural design decisions. We will also show how this iterative process changes the nature of architectural design decisions as the system matures.

4.2.1 Tasks as motivators

Architectural design decisions are made by the developer that is responsible for implementing the decision as described by 4.1. Exceptions to this rule are technological selections which are always made by the architect.

Assigned to: Dev 2	task: 28
Task description: Customer management in Ekukka shop. The idea is that all Ekukka shops have a common customer base (which is handled by ekukka.fi). Single page login etc.	

Table 4.1: Example of a task assigned to a developer

The starting point for an architectural design decision is a development task that implicitly contains an architectural design decision. It is the need to completing the task that motivates a developer to make an architectural design decision. For example task number 28 described in 4.2.1 is assigned to DEV 2 (top right in the diagram) and contains an architectural design decisions: How should the end customer base system be integrated into Ekukka shop system so that a customer that has registered into Ekukka end customer system will be able to login in any Ekukka shop.

The development team has no implicit process for how design decisions should be made. In practice design decision is made by the individual developer that is responsible for implementing the decisions - or in informal discussions between members of the development team (appendix, interviews, Dev 1). These discussions sometimes also involve the product owner as described in the previous section.

Since one of the developers mentioned that only major decisions are made with the team (interviews, Dev 1) and this has been the authors observation as well - it is likely that vast majority of the decisions are made by the individual developers and implemented without going through any review. We will later discuss the problems of this method of operation, however it was pointed out by the architect that it also carries the benefit of making the developers responsible for their implementations since they cannot blame the architect for bad designs (appendix, interviews, SM).

4.2.2 Role of product owner

Since the product owner is the holder of requirements in the organization the product owners role in making architectural design decisions is key. There are essentially two ways the product owner can be involved in: as an active participant in the decision making or as a passive source of rational.

Decisions that the PO has an active role in are those where the product owner comes up with a design that at least partially solves the question - typically a user interface drawing. In these cases the development team is left with filling potential blanks in the design and coming up with the technical implementation of the design. Design decisions of this type are typically

limited to matters concerning the layout of the user interface.

Since the developer has the ultimate responsibility on how the design should be implemented into the system we do not consider the PO to have made an architectural design decision even when he does come up with a plan for the feature. It is up to the developer who implements the design to take care that the implementation does not have an adverse effect on the systems architecture.

The second type of decisions from the PO‘s perspective are design decisions where the PO is consulted for the rational of the decision. These are typically decisions of a more functional nature. Often the question is how some feature should work to best satisfy the demands of the users or our internal business logic - both of whom the PO is expert of. For example the author recently inquired the Product Owner on whatever the shopkeepers should be able to see an order that the end customer has not paid yet.

There are of course design decisions that do not directly involve the product owner. These are typically decisions where the quality attributes in question are of purely technical nature, such as evolvability, reliability or scalability of the software. Of course even in these questions the developer should be aware of the business logic of the system being developed - in 4.4 we will see an example where a failure to understand the business logic of the system lead to a developer missing the critical aspect of scalability.

4.2.3 Identification

As we discussed at start of this chapter architectural design decisions are a subset of design decisions that are architecturally significant. Since they tend to have a fundamental effect on the system that is being developed it is assumed that more effort is spent in getting architectural design decisions right than is spent on more routine questions. Thus it is very important that the developers correctly identify architectural design decisions as architecturally significant.

The development team does not have an explicit strategy for identifying architectural design decisions, with the exception of aforementioned technology choices which the architect always has final say on. In fact it would appear that the development team shares no explisit definition for what is an architectural design decisions - rather in the interviews developers preferred using terms like critical decisions when talking about architectural design decisions.

From the interviews it is clear that different developers have different strategies for identifying what they deem to be critical decisions. For example the third developer gave an example of a decision that vastly affected the ease

of testing and further development of the software, matching the idea that the effect on quality attributes makes decision architecturally significant. The first developer on the other hand though that decisions that lay a foundation for the software are critical, which matches with the idea that decisions that are hard to reverse once implemented are architecturally significant.

4.2.4 Sharing the responsibility

As described in 4.2.1 the responsibility of making decisions on how to implement a feature or subsystem falls to the developer responsible for its implementation with the exception of technology selections which the architect has responsibility for. For example the decision on how to implement end-customer login in Ekukka shop falls to DEV 2 since the task of implementing the system has been assigned to.

The team member may decide to share the responsibility of making an architecturally significant design decision with rest of the development team, how ever doing so is up to the individual developer since he has the ultimate responsibility on the decision. From the interviews and observations there appear to be three factors that affect the developers willingness to share the architectural issue with others:

1. Does the developer view the decision as architecturally significant ?
2. Does the decision affect components that are taken care of by other developers ?
3. Does the developer have relevant domain knowledge on the field of the decision ?

We have already discussed the strategies for identifying architectural design decision in the previous section. The effect of the component ownership is demonstrated by a quote from the first developer: “If it is say a UI side script problem, then there is not much point harassing others and I will rather solve it myself”. Figure 4.3 illustrates who is responsible for what in the development of the end customer web shop module and how this affects the division of responsibility. We can see that each component of the system tends to be under responsibility of a certain developer, however DEV 1 and 2 share responsibility for Customer Web Shop as a whole and ARCH and DEV 3 have responsibility for the interfaces with image gallery and customer registry. These ownerships are very informal and the various components can and do change hands from time to time - for example the image gallery has been handed over between the ARCH and DEV 3 a couple of times already.

No rule prevents someone else from modifying a component belonging to another developer, however there is an understanding between developers that the structure of the module is up to the developer in charge of it.

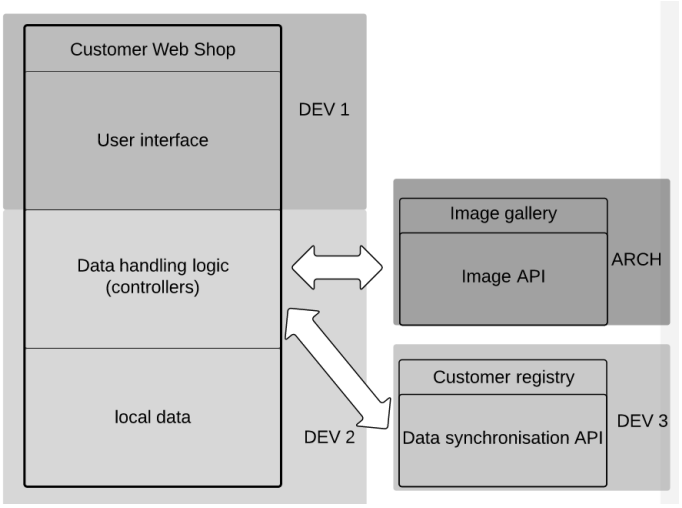

Figure 4.3: Areas of expertise.

In practice for the Customer Web Shop the division of labor means that DEV 1 can make decisions that concern how things are displayed on the user interface - but a change that concerns what data is shown requires attention from DEV 2 who is responsible for the data handling logic. Correspondingly DEV 2 can independently make decisions on how data is stored - but needs DEV 1‘s attention if he wants to change how the data is displayed to end users. In similar manner the ARCH and DEV 3 can independently make decisions on the internal structure of the customer registry and image gallery - but have to consult DEV 2 if the interface between these components and the customer shop needs to be changed. Naturally decisions with a large scope tend to overlap with many different components and thus get shared more often than decisions with smaller scope.

The third major factor on whether to share the decision is the decision maker‘s knowledge of the domain the question falls under. This also is visible in the start of the first developers comment: “If it is say a UI side script problem then there is not much reason to bother others and I will do it myself”. Since the first developer has by far the most experience on the team with working on web user interfaces it is natural that he feels little reason to consult other developers on such issues. On the other hand other developers are likely to consult him if they have a UI side decision to make. Another

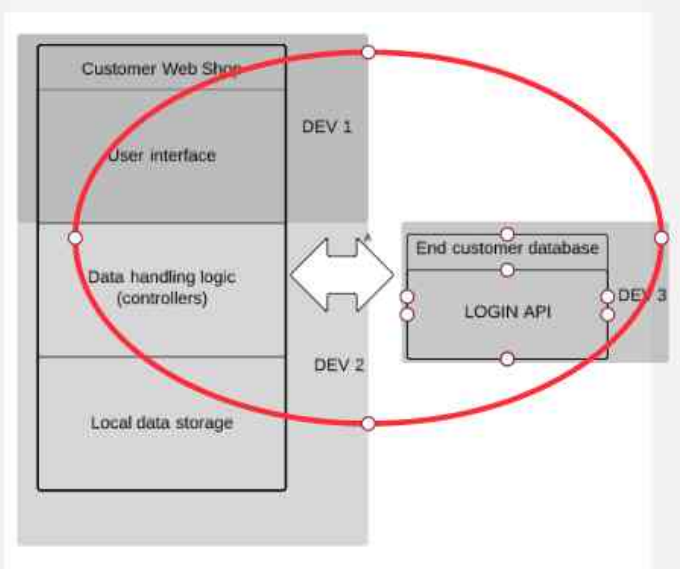

Figure 4.4: Effect of architectural decision on how to implement centralized login marked on red.

example of domain expertise based decisions occurred early in the project, when I designed the data model for the web shop. Although as we can see from figure 4.3 that developing the data model is my sole responsibility and I have considerable experience from handling systems, I haven't got much experience in developing large scale systems. The architect on the other hand has and thus we went through the data system together and came up with a solution that was fundamentally different from what I originally proposed - going with a system with multiple individual databases rather than a single database.

To summarize there are three major factors that affect developers decision to share the decision: whether the developer views the decision as architecturally significiant, whether the developer thinks the issue will affect systems that are responsibility of other developers, and finally whether the developer thinks he has enough relevant domain knowledge. For example let us consider all of these aspects in the previously presented 4.1.2 design decision of how to implement the single system login in Ekukka shop system:

1. Criticality: Very high. Although the shop system can be used without registering into shop, a working system for logging into the shop system is critical since it allows customers to make orders without having to enter their address information every time. The safe and proper han-

dling of customer information is very important part of maintaining customer relations - a single mishap can cause a massive set back.

2. Does the decision affect components that other developers are responsible for: Yes as described in 4.4. There is certainly a need to consult with developer 3 over how the end customer registry should provision logins as represented by the diamond in 4.4 . Developer 1 should be consulted over how the login should be integrated in the shops user interface as described by overlaid area in 4.4.

3. Domain knowledge: Because security is such a high quality factor in this decision the architect should be consulted over the matter since he has most domain knowledge on security issues.

Conclusion: Since the decision affects all of the developers and requires attention from the architect the whole development team should look into the matter.

Because there is little managerial support the quality of the architectural decision itself relies heavily on the experience and competence of the developers in identifying critical decisions as well as making the right decision. Critical is also the developers ability to understand what the system they are building should do. This is concurrent with the agile emphasis on the role of developers in creating systems as discussed in agile architecture.

When developers do find an item critical enough to share with other developer, the decisions on the item are usually done informally either in face to face discussions, online meetings or via Flowdock. The developers and architect prefer to do the decisions in face to face discussions, with the architect emphasizing the role of the team working together in the same room. It has also been the writers experience that most decisions get made between the team when they work with the system, not in separate planning meetings - and the decision making progress is most effective the developers work in the same space. We will return to this topic in chapter 4.5.

4.2.5 Project phases

The interviews made it clear that the development team has no formal time frame for making architectural design decisions. However observations made during development and the interviews with the developers allow us to identify three development phases were identified in relation to architectural design decisions: planning period, foundation period and iterative development period.

The planning occurs at the start of the project, and consists of the architect and developers going through the requirements of the system with the product owner. During this phase the team chooses what technologies will be used within the project and how the system integrates with other system. Depending on the projects complexity, the team may also decide how the data should be structured and some architectural pattern for structuring the code. For example with the end customer web shop the developers created a graphical representation of the relational database that would be used. The architect then adjusted this model to better facilitate scaling the project. Indeed since this part of the development occurs before actual coding, it tends to create the largest amount of drawings and other design artifacts. The length of the planning phase is also heavily dependent on the projects complexity, typically lasting from two days to two weeks.

The foundation period occurs after the planning phase has concluded and the team has made the high level design decisions. During this time the developers and the architect will start coding the system itself, making decisions on how the code should be structured into components and how the components should talk with one another. This phase typically lasts from a week to a month, until the first version with all the necessary structures in place is finished. Typically this phase does not involve the product owner as much, since the decisions made are of very technical nature.

After the foundation period the development moves to more cyclical development, with the team coding some features in, responding to feedback from the product owner and then adjusting the features based on this feedback. At this phase there appear to be three typical triggers for architectural decisions. The first is that the requirements set for the system change - typically when the product owner comes up with a new feature or bring a new requirement for an existing feature. The second is that the product owner notices that the current system does not meet its requirements and requires a change. The third is that the developer notices that the system no longer meets the requirements set on it. It should be noted that in all cases the requirements can be functional such as: the users must be able to change their passwords or non-functional such as: the system should be easier to test to allow faster development.

Estimating the exact numbers of architectural design decisions made during the various phases is difficult, however estimating the relevant numbers is quite possible. From the observations made during the development it appears that speed with which developers can make design decisions is dependent upon the complexity of these decisions. However the complexity of the design decisions tends to correlate with the effectiveness of the decisions, that is how much the decision affects the quality attributes of the system and

how difficult it is to reverse. Thus the more complex design decisions are the more likely they are to be architecturally significant.

The author has observed that the amount of design decisions is highest at the cyclical development phase, since at this point the team implements and refractors a large number of functions every week. This is because the foundation for the system has been set, and the design decisions tend to be of very low complexity, mostly routine. These routine decisions tend to have very little effect on the structure of the system - leading to the actual amount of architectural design decisions made in iterative development being relatively low.

On the other hand at the planning phase the team typically only tackles a small handful of design decisions, but since they are all high level decisions they tend to be of high complexity and have great effect on the system, thus making almost all of them architecturally significant. During the foundation period the team can already tackle a far larger number of design decisions, and although they do not tend to be as high level as in the planning phase, they still have a great effect on the system being developed and thus tend to almost all be architecturally significant. Thus the highest number of architectural design decisions is made during the foundation period.

4.3 Communicating architectural design decisions

For the developers to make good architectural decisions they must be aware of the constraints presented by previous decisions. Thus the communication of decisions that have been made is vital for smooth running of development.

From the interviews as well as authors observations working with the team, three forms of communication are used for sharing architectural design decisions: verbal, document driven and system driven.

4.3.1 Verbal communication

The most active communication method used by the team. Verbal communication of the architectural decisions that have been made is mostly done in face to face discussions but also in on-line meetings. Verbal communication occurs both in an active manner with the developer sharing the decision he has made with other members of the development team - as well in a passive manner when one of the other developers wants to know how a feature has been implemented.

Verbal communication of the architectural design decisions that have been made tends to be very informal - most often right after the developer has implemented the feature or decided on how the feature should be implemented.

The advantage of verbal communication is its ability to share not only the effects of the decision - but also the rationale behind it. Indeed sharing and discussing the rationale with other developers can occasionally expose a weakness in an architectural decision that has been made and allows the team to reconsider the decision.

The weakness of verbal communication is its extreme temporality since developers tend to forget information they have received and verbal communication does not leave any permanent records.

4.3.2 Document based communication

The use of architectural descriptions varies from person to person and according to the situation, but it should be noted that no comprehensive architectural documentation is used. Earlier the team attempted to have a more comprehensive development documentation in form of a wiki, however it rapidly fell out of use. After a long period of inactivity the database for wiki for accidentally written over, and the team decided to drop it completely. Instead development team uses a wide variety of architectural artifacts to describe some aspects of the system.

In addition to lacking comprehensiveness the document tends not to contain much of the rationale for the decision but rather focus on describing their effect on the structure or running of the system. It is of course possible to grasp some of the rationale from the way the system works or is structured especially with some of the more comprehensive artifacts.

Artifacts vary strongly when it comes to their temporality from descriptions that only meant to be used for a single meeting to descriptions that are meant to help the development work for the whole duration of the project. While working with the team the author has identified three commonly used artifact types:

Throw away descriptions: Artifacts used to describe ideas and concepts, typically in meetings but also occasionally when developers discuss different approaches. Almost always drawn by hand, typically on flip board sheets. Despite the fact that throw away artifacts are mainly used as a visual aid in meeting environments, they are sometimes kept around for a while as records on what was decided or even used for implementation.

Snapshot descriptions: Artifacts used to create and describe architectural designs. Most often used when a developer requires an interface from a system that is under being developed by another developer. Snapshot designs

are typically more comprehensive than throw away artifacts and more often than not follow an established convention for design description, thus making them easier to understand. Snapshot designs are not updated after their creation, which causes them to go obsolete.

Long term descriptions: Artifacts used to create and describe architectural designs. Unlike snapshot descriptions, long term designs are updated as the design changes. Long term descriptions are typically used when the developer wants to design the system separately from its implementation. Unlike throw away and snapshot descriptions, long term descriptions tend to be digital. This makes changing the design easier than when dealing with paper descriptions.

4.3.3 System based communication

Since there is no comprehensive documentation of architectural decisions available the developers and the product owner have to occasionally rely on looking at the system to understand architectural design decisions. This way it is often possible to understand the effects of an architectural design decision - but grasping the rationale behind the decision is even more difficult than with architectural documents. The advantage of looking at the system for architectural knowledge is that it is up to date.

There are essentially two ways to gauge information from the system. First is to look how the system works from users perspective. This is the way naturally preferred by the product owner and allows one to see how well the design of the system matches the needs of the user. The other way is to look at the systems code. On principle this allows full visibility into how the system has been implemented, however attempting to understand code that is not ones own is very difficult and exhaustive work. This process is made easier if the coder has used standard structures in his work as well comments on how the code should work.

4.4 Pitfalls of architectural design decision making

From the interviews, personal experience when working with the team and observations the author has identified two major problems that have occurred during the development

A: Developers occasionally make bad architectural design decisions. B: The developers and product owner do not always share a common vision on how the system should work.

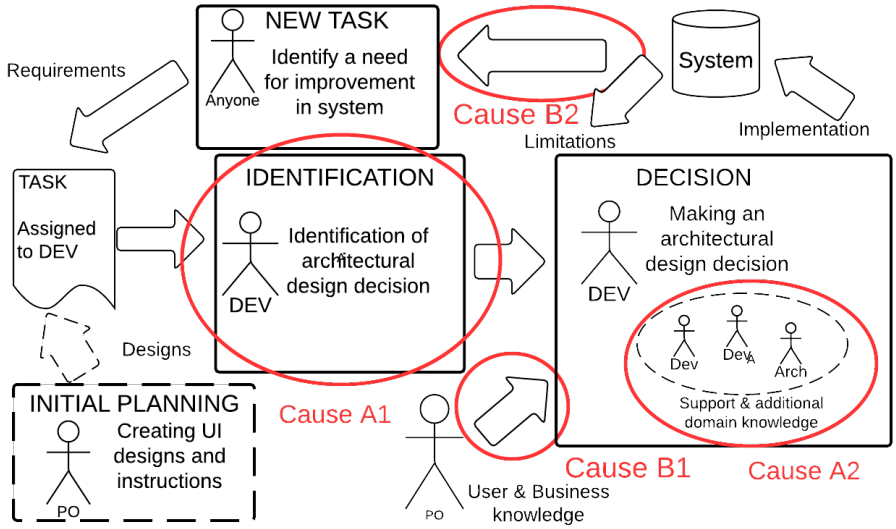

Figure 4.5: Problems with the way architectural design decissions are made.

As can be seen from figure 4.5 problem A is related to the way architectural design decisions are identified and made and problem B is related to communication between the product owner and the team in the decision making and visibility of the system effected by the architectural design decision.

There are two major sources for problem A.

A1 Developer that makes an architectural design decision fails to identify the decision as architecturally significant.

A2 Developer correctly identifies an architectural design decision, but chooses a bad solution.

The first problem (A1) occurs when developer that makes an architectural design decision fails to identify this decision as architecturally significant. As pointed out by [30] identifying architectural design decisions is often very difficult, since even small scale decisions made can have unforeseen consequences. Not identifying the decision as critical means that the developer is likely to pick whichever solution comes to mind first, without considering the consequences of the decision or the alternatives to that decision. As discussed before it is also highly unlikely that a developer will share with others a decision that they find routine.

An example of a decision not identified as critical comes from when we created a reporting module for our business to business web shop. The business to business web shop runs on an open source web shop platform that is very customizable, but also fairly complex and heavy. On a high abstraction level it has five tiers, a database, a class based system for abstracting database objects into class based objects, a controller system for handling data management, a template system for displaying data from controllers to UI and finally a UI layer for user interface side functions and visual styling of the user interface.

When we started creating a reporting tool for the system we had only one type of report that we had to get running quickly, and thus we did not give much consideration to expandability of the system. We were also still in development, and thus had little experience working with heavier data sets. Thus the decision on how to retrieve the data did not seem terribly critical to me, and I decided to do it as we had been doing with other systems we had developed, use the web shops own class modules and controllers to retrieve and compose the data.

Unfortunately this decision turned out to be the wrong one. As we went into production the number of orders rose very rapidly and after about a year the system was unable to reliably make reports for even a weeks worth of orders - simply running out of memory in middle of the progress. To compound the error five other reports had been built on the same system, which now turned out to be unusable. The solution was simple - bypass the class based system and retrieve the data needed directly from the database - which was easily able to handle such queries. About ten days of man hours were wasted in having to implement the system twice - a costly error in middle of busy development. The primary reason for this error was not understanding the scalability of the web platform system.

The second problem case (A2) occurs when developer identifies the issue as critical, but ends up making a bad decision regardless. In these cases the developer usually fails to notice one the consequences of the decisions he makes, such as how it affects the quality factors of the system. This problem is usually caused by lack of inter-team communication, since the developer has already identified the problem as critical and should be willing to share it. The lack of sharing can either be because the developer views the sharing as too inconvenient, or because the other developers are too busy with their own work and not responsive to developers attempts to get help with making the decision.

Triggering factor is that the developer does not recognize the decision as being critical enough to cause major problems down the road - or that the developer tries to bring the problem to the attention of the rest of the team

but the team is too busy to take it into serious consideration - thus leaving the issue on the shoulders of the single developer. It is of course possible for a group of developers to make a bad decision as well - however considerably less likely.

Problem B also has two primary causes.

B1 Product owner is unable to monitor the development of the system.

B2 Communication breakdown between the development team and product owner.

B1 occurs when the product owner is unable to monitor the status of the system under development. Since the instructions coming from the product owner are usually not very exact and often need to be analyzed by the developer the system tends to divert from the product owners original vision. When the product owner is constantly able to inspect the system he is able to request changes that will set the system back on track with fairly light amount of effort. However when this does not occur the developer will start to build on the earlier deviations making it exceedingly difficult to get the system back on track. The lack of access to the system may also make it more difficult for other developers to inspect the system for defects or to offer help in making of architectural design decisions, contributing to problem A.

That the team had fallen into this pitfall became apparent during the writing process of this work. The third developer had been working on the customer database for over a year but due to communication mishaps no staking server had been set up for the project. This meant that there was no place for the product owner to view the work done by the developer. Since the communication between the developers and the product owner was sporadic at best due to the product owners physical distance from the development team, their vision on the system could diverge for months at a time. The ensuing cycle of occasional attempts to realign POs vision with the existing system only for them to diverge again lead to a system that in the end had a patchwork of bad design decisions piled upon one another.

The second problem case (B2) occurs when developers and product owners have difficulties communicating with one another. It is a more high level problem but one that does clearly rise from the interviews and especially from in informal discussions with developers. Since the PO not only works part time but is based almost 600 kilometers from rest of the development team it can occasionally become difficult for the developers to reach the PO when they need information on the system they are building. This means that the developers have to guess what the PO would want - diverging the system from the PO‘s vision.

This problem rose to prominence during the construction of the Ekukka shop system when the product owner became involved in a major project in his day job and was unable to take as active part in the development of the system as usual. At that point the admin side of the system had already been designed and partially implemented, but the customer side only had a very rough layout planned. Since delaying the work was not an option and the Product Owner was not available, the developers had to design the user interface based on best estimates and guesses. This lead to great divergence between the PO's vision and how the customer side was actually implemented. Some architectural design decisions were also compromised due to the need to guess how the system should work.

4.5 Recommendations for solving the problems

In the previous section we presented two problems that are caused by the way architectural design decisions are made in the company:

A Developers occasionally make bad architectural design decisions.

B The developers and product owner do not always share a common vision on how the system should work.

We identified two direct causes for the first problem (A):

A1 Developer that makes an architectural design decision fails to identify the decision as architecturally significant.

A2 Developer correctly identifies an architectural design decision, but chooses a bad solution.

Based on the interviews, discussions with team members and experiences working with the team the author proposes two solutions for the causes of problem A:

A:S1 Introductions of weekly meetings between developers to facilitate the identification, sharing and documentation of architectural design decisions.

A:S2 Facilitation for more co-located development to facilitate sharing of architectural design decisions.

The first solution (A:S1) was favored by the author and the scrum master, while the architect found the solution beneficial but probably inadequate. The purpose of this solution was to get the developers to share the decisions they had made as well as decisions they had problems with and needed help. The purpose was also to find a way to document at least the most critical of these decisions.

The second solution (A:S2) was favored by the architect and supported by the interviews in which all interviewees except the product owner stated that they preferred face to face meetings over online or written communication. The purpose of this solution is to help developers share the decisions they find architecturally significant as well as those they think might be architecturally significant. Based on observations made during the project the author believes that the developers are much more willing to share architectural design decisions when they are in face to face contact with other members of the development team.

We also identified two primary causes for the second problem (B):

B1 Product owner is unable to monitor the development of the system.

B2 Communication breakdown between the development team and product owner.

As with the first problem (A) we also two present two solutions for the second problem (B):

B:S1 Introductions of monthly meetings with all team members present where the current status of the project can be reviewed.

B:S2 All development work should be continuously visible to product owner as well as rest of the team.

The goal of first solution (B:S1) is to alleviate the second cause (B2) by ensuring that developers have access to the product owner at least once a month. From the interviews as well as discussions with developers it became apparent that developers often feel that their access to the PO is limited when the PO is busy with other projects. As was the case with other developers it is also apparent that developers have a higher threshold for querying the PO for information when the PO is not locally accessible. On the long run developers making architectural design decisions on bad information can lead into serious architectural problems.

The goal of the second solution (B:S2) is to ensure that the PO always has the possibility to inspect the work the developers are doing and so to prevent the second cause for diverging visions (B2). As described in 4.3.3 the system

under development is the only place where the effects of design decisions are certainly visible. When the PO is able to see the effect of decisions on system under development he can steer the developers back to the right path ensuring that no further decisions are made based on bad decisions.

These two solutions should also support one another. Naturally there is no point in ensuring PO visibility into systems under development if the PO does not monitor these systems - the monthly meetings should allow the developers to present their work to the PO thus ensuring that PO is aware of the development being made. At the same time ensuring that PO is able to see the systems makes the monthly meetings, as well as meetings in general more fruitful since the PO is on the same page as the developers.

Chapter 5

Discussion

5.1 Answers to research questions

We presented four research questions at start of the work. Based on the results presented in the previous chapter, we will now answer these questions.

How are architectural design decisions made?

Based on the results presented in 4.2 we can answer four central aspects of this question:

1. Who make architectural design decisions ?

Architecturally significant design decision are made by the development team with the help of the product owner. The responsibility for the technical aspects of the architecturally significant design decision falls to the developer that is responsible for implementing the decision.

2. How are architectural design decisions identified?

Developers do not explicitly identify design decisions as architecturally significant but do identify some decisions as being critical. Developers base this identification both on how much the decision effects the quality factors of the system as well as how difficult the decision is to reverse in the future.

3. In what cases do developers share the responsibility for the architectural design decision?

Developer may decide to share the architecturally decision with other developers or the architect. There appear to be three factors that effect this decision: 1. Does the developer views the issue as critical, 2 Does

the developer thinks the issue will affect systems that are responsibility of other developers, 3. Does the developer think he has enough relevant domain knowledge. Additionally the developers decision is effected by how easy it is to reach the other members of the development team.

4. In which phases of the project are architecturally significant design decision made in

Project appear to have three distinct phases when it comes to making of architectural design decisions: Planning, founding and iterative. Very high level decisions are made in planning phase. Decisions that critically affect the structure of the system are made in the founding level. Finally in the iterative development phase the decisions become routine enough that few of them are architecturally significant.

How are the decisions that have been made communicated?

Architectural design decisions are not explicitly recorded but they are some times shared verbally. Artifacts that detail the relevant architecture are often used when decisions are shared between developers, however the rationale behind the decision is almost never recorded and can only be observed implicitly. In the common case where no separate architectural descriptions are made, the only recording of the architectural decision is the system that it affected.

What problems arise from the current architectural practices?

Two problems arise from the the way architectural design decisions are made.

First is that the product owner and developers do not have a common vision on how the system should work. There are two causes for this: The inability of the developer to access the product owner and inability of the product owner to monitor the system under development.

The second problem is that developers make bad architectural design decision. There are two primary causes for this: First is that a developer does not correctly identify an architectural design decision as architecturally significant. Second is that although a developer has identified the architectural design decision correctly he ends up making a bad decision.

How could these problems be solved?

Two recommendations were presented for solving the lack of common vision between developers and product owner. First is to have monthly meetings with all team members present where the current status of the project can be reviewed. This would not replace direct interaction between developers and the product owner, but rather act as a safety mechanism for making sure that the developers and the product owner are on the same page. Second recommendation is to mandate that all development work should be continuously visible to product owner as well as rest of the team.

Two recommendations are also presented for solving the problem with developers. First is to have weekly developer meetings with developers and architect present. In these meetings the developers can report how they are doing and if they have come across any architecturally significant decision they need to make. As with the monthly meetings, this is not to replace direct interaction between developers and the architect, but rather to act as a safety that keeps the architect as well as other developers on track as to what the developer is doing. It also allows an environment where developer can easily ask for help with an architecturally significant design decision. The second recommendation is to increase the share of development time the developers and architect work face to face - since developers find it easiest to share decisions when they are working together.

5.2 Reflections with literature

The findings made with this case are fairly well in line with what agile literature and research into agile say of software architecture. The team has adapted well to the agile principle of 'The best architectures, requirements, and designs emerge from self-organizing teams" [6] with no hierarchical roles in how architectural design decisions should be made. The architect has retained some of the architectural tasks such trying to ensure overall architectural quality and ensuring that refactoring of code is done when necessary - but other tasks such as making of architectural design decisions is delegated to team members. This is in line with extreme programming - which allows architectural tasks to be divided with the team. [5] It is also in line with Scrum, which does not explicitly define architect role but rather considers it as one of the optional developer roles which can be filled as required [27].

The findings on when architectural design decisions are made is in line with agile literature, however not in line with any specific agile methodology. Rather it seems to be in between of the extreme programming principle of

making only the absolutely necessary decisions [5] and Crystal and DSDM attitude of doing some architectural planning beforehand [7]. Since the team does not follow any specific methodology when it comes to planning, each developer has to decide individually how much they want to plan ahead in their architectural design decisions.

The observations that the amount of architectural design decisions drops as the development goes on is in line with agile thinking. The idea that making the system flexible so it can adjust to changes in requirements without needing major restructuring is highlighted in extreme programming [5]. Martin Fowler also concluded that one of agile movements key ideas is to make architecture unnecessary by designing the system so that all decisions are reversible [15] - although this approach itself requires attention to architecture.

It is logical yet slightly ironical that the end goal of both plan driven methods and agile methods appears to be the same. After all the plan driven methods attempt to make the development work as smooth as possible by making the complex decisions at the start of the project [7] while agile methods attempt to make development as smooth as possible by reducing such complexity. It should also be noted that by doing some planning beforehand the team in this case takes advantage of plan driven methods to allow for smoother development. It should be also noted that this is in line with some agile methods such and indeed is what some authors such as Boehm [7] recommend teams do.

The problems discovered stem at least somewhat from the ways the team fails to follow the agile values. Following the value 'Individuals and interactions over Processes and tools" suffers from the fact that the team is not fully collocated and thus team members have fewer chances to freely interact with one another - instead relying on communication tools and scheduled meetings. This contributes both to the problem of developers not sharing difficult architectural design decisions and staying on the same page with the product owner.

The solving of this problem is less than trivial. Although this work suggests increasing the amount of co-located development, there is no escaping the fact that distributed software development is becoming more and more common and something even agile teams cannot escape [22].

The more critical problem with keeping the product owner on the same page occurred when the team failed to adhere to the value of Customer collaboration over Contract negotiation. Since the product owner represents the customers in a scrum project, the fact that for a very long time the product owner was unable to see what was happening with one principle system not only caused serious problems but was also directly against agile

values.

5.3 Reflections in quality of research

Conducted over a period of nine months, this research had a good opportunity to observe and participate in one agile teams road from start of a project to the first public release. Observations that were made as well as the interviews that were conducted gave in the authors' opinion solid ground upon which to build the conclusions made in this chapter.

However that is not to say that this research process has been without its fault. The biggest of these is that it has fallen into the trap mentioned in methodology chapter - namely that it has done too much research with too little action. Most of the suggestions made for fixing the teams problems have not been effective due to scheduling as well as team centric problems. Although the latter problems give us a look into the agile teams reluctance to change the way it does things - as well into the difficulties of trying to solve problems of distributed agile teams - there is no escaping the fact that more successfully taking action into these problems would have given us more knowledge of how agile developers actually do architectural design decisions.

To prevent the problem mentioned above this research used cyclical research model. However the pressure of doing the research while working on the software with the team lead to some parts of the research method being neglected. The structure of cyclical action research is to do diagnosing, action planning, action taking, evaluating, and reflection phases in a cyclical manner. The plan was to do two of these cycles. However in practice only one cycle was fully completed - with a partial cycle introduced midway to adjust the course of the research.

In the writer's opinion this shortcoming was caused by two factors. First the writer failed to get the team fully on board with the research. Although some of the management was fully on board and even developers agreed that something should probably be done to improve the development process, no proper conclusion was reached with different parties on what that something should be. This problem was heavily compounded by the fact that it was very difficult to get the same parties together for any extended period of time so this matter could have been discussed and properly concluded. These issues led to the attemps at introducing solutions being at best half effective.

Second problem was that the first diagnosis phase took the writer longer than expected. This was probably somewhat inevitable considering the master thesis work has a large theoretical part which needs to be written. Combined with doing the interviews and observations that were necessary for

finding the teams problems, the writer was ready to move on to action planning and action phases a month later than planned. Since the results of these parts as well as the diagnosis also had to be written up not enough time remained to give enough attention for the action itself.

Chapter 6

Conclusions

Allowing developers to make architectural design decisions independently does make the development process more straight forward and carries many benefits. However it demands much of the developers ability to both identify and make architectural design decisions. Organisations should ensure that developers have easy access to support from an architect or other developers when their experience or domain knowledge fails them.

Scrum organisations should also take into account that when making architectural design decisions developers rely heavily on the product owner for the latters understanding into how the system should work. At the same time the Product Owner needs access to the system under development so he can observe what the developers are doing.

Both of these problems seem especially concerning for distributed teams. Indeed it is the authors opinion that futher understanding into these problems would be gained by studying both agile teams that are fully colocated, as well as by studying teams that are divided into multisite teams.

Bibliography

[1] Avison, D. E., Lau, F., Myers, M. D., and Nielsen, P. A. Action research. *Communications of the ACM 42*, 1 (1999), 94–97.

[2] Bannerman, P. L. Software architecture: Organizational perspectives. In *Proceedings of the 2009 ICSE Workshop on Leadership and Management in Software Architecture* (2009), IEEE Computer Society, pp. 37–42.

[3] Baskerville, R. L., and Wood-Harper, A. T. A critical perspective on action research as a method for information systems research. *Journal of Information Technology 11*, 3 (1996), 235–246.

[4] Beck, K. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[5] Beck, K. *Extreme programming explained: embrace change*, second ed. Addison-Wesley Professional, 2004.

[6] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. The agile manifesto, 2001.

[7] Boehm, B. Get ready for agile methods, with care. *Computer 35*, 1 (2002), 64–69.

[8] Clements, P., Ivers, J., Little, R., Nord, R., and Stafford, J. Documenting software architectures in an agile world. Tech. rep., DTIC Document, 2003.

[9] Clements, P. C. *Software architecture in practice*. PhD thesis, Carnegie Mellon University, 2002.

[10] Cohen, D., Lindvall, M., and Costa, P. An introduction to agile methods. *Advances in computers 62* (2004), 1–66.

[11] Cohen, L., Manion, L., and Morrsion, K. *Research methods in education*, second ed. Croom-Helm, Dover, NH, 1980.

[12] Davison, R., Martinsons, M. G., and Kock, N. Principles of canonical action research. *Information systems journal 14*, 1 (2004), 65–86.

[13] Dickens, L., and Watkins, K. Action research: rethinking lewin. *Management Learning 30*, 2 (1999), 127–140.

[14] Farenhorst, R., and de Boer, R. C. Architectural knowledge management: Supporting architects and auditors. *VU University* (2009).

[15] Fowler, M. Who needs an architect? *IEEE Software 20*, 5 (2003), 11–13.

[16] Hofmeister, C., Nord, R., and Soni, D. *Applied software architecture*. Addison-Wesley Professional, 2000.

[17] ISO/IEC/IEEE 42010 Systems and software engineering ? Architecture description, 2011.

[18] Iversen, J. H., Mathiassen, L., and Nielsen, P. A. Managing risk in software process improvement: an action research approach. *Mis Quarterly* (2004), 395–433.

[19] Jansen, A., and Bosch, J. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on* (2005), IEEE, pp. 109–120.

[20] Jansen, A., Bosch, J., and Avgeriou, P. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software 81*, 4 (2008), 536–557.

[21] Jansen, A. G. J. *Architectural design decisions*. PhD thesis, University of Groningen, 2008.

[22] Korkala, M., and Abrahamsson, P. Communication in distributed agile development: A case study. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on* (Aug 2007), pp. 203–210.

[23] Kruchten, P. Software architecture and agile software development: a clash of two cultures? In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* (2010), vol. 2, IEEE, pp. 497–498.

[24] Kruchten, P., Obbink, H., and Stafford, J. The past, present, and future for software architecture. *Software, IEEE 23*, 2 (2006), 22–30.

[25] Kruchten, P. B. The 4+ 1 view model of architecture. *Software, IEEE 12*, 6 (1995), 42–50.

[26] Malan, R., and Bredemeyer, D. Less is more with minimalist architecture. *IT Professional 4*, 5 (Sep 2002), 48, 46–47.

[27] Mundra, A., Misra, S., and Dhawale, C. Practical scrum-scrum team: Way to produce successful and quality software. In *Computational Science and Its Applications (ICCSA), 2013 13th International Conference on* (June 2013), pp. 119–123.

[28] Nord, R., and Tomayko, J. E. Software architecture-centric methods and agile development. *Software, IEEE 23*, 2 (March 2006), 47–53.

[29] Poort, E. Driving agile architecting with cost and risk. *Software, IEEE 31*, 5 (Sept 2014), 20–23.

[30] Poort, E. R., and van Vliet, H. Architecting as a risk-and cost management discipline. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on* (2011), IEEE, pp. 2–11.

[31] Ramesh, B., Cao, L., Mohan, K., and Xu, P. Can distributed software development be agile? *Communications of the ACM 49*, 10 (2006), 41–46.

[32] Reifer, D. Xp and the cmm. *Software, IEEE 20*, 3 (May 2003), 14–15.

[33] Robey, D., and Markus, M. L. Beyond rigor and relevance: producing consumable research about information systems. *Information Resources Management Journal (IRMJ) 11*, 1 (1998), 7–16.

[34] Schwaber, K. *Agile project management with Scrum*. Microsoft Press, 2004.

[35] Shahin, M., Liang, P., and Khayyambashi, M.-R. Architectural design decision: Existing models and tools. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on* (2009), IEEE, pp. 293–296.

[36] Stettina, C. J., Heijstek, W., and Fægri, T. E. Documentation work in agile teams: the role of documentation formalism in achieving a sustainable practice. In *Agile Conference (AGILE), 2012* (2012), IEEE, pp. 31–40.

[37] Susman, G. I., and Evered, R. D. An assessment of the scientific merits of action research. *Administrative science quarterly* (1978), 582–603.

Appendix A

Interview questions

Taustaa

-Haastateltavan koulutustausta (viimeinen / korkein suoritettu tutkinto / meneillään olevat opinnot).

- Haastateltavan tausta yrityksen toimialalta (floristiikka, vähittäismyynti, tukkumyynti). Onko työskennellyt alalla / opiskellut alaa / kiinnostunut alasta jne.

-Haastateltavan tausta ohjelmistotuotannon puolelta (eli suomeksi onko aikaisempaa kokemusta it projekteista etc).

Haastateltava osana kehitysryhmää

-Haastateltavan rooli kehitysryhmässä (haastateltavan omasta mielestä). Mitä työtehtäviä haastateltava hoitaa / mihin on erikoistunut etc. Tarvittaessa esimerkkejä tyypillisistä työtehtävistä.

-Millaisia muita rooleja haastateltava näkee kehitysryhmässä? Miten nämä näkyvät käytännössä ?

Ohjelmistokehitysprosessi, tehtävät

Mistä / keneltä kehitystehtävät tulevat, miten ne jaetaan kehittäjille ja miten niiden edistymistä seurataan?

Mitä tapahtuu valmiille tehtäville, ja miten niiden valmistuminen määritellään ?

- Mikä on kehittäjän oma rooli ohjelmistokehityksessä ?

Ohjelmistokehitysprosessi, vaatimukset

- Ohjelmistokehityksessä tehtäville tehtäville asetetaan yleensä joitain vaatimuksia. Miten nämä vaatimukset näkyvät osana tehtäviä ? Miten kehittäjä käyttää vaatimuksia tehtävän toteuttamisen aikana ?

-Millaisessa muodossa vaatimukset yleensä tulevat ? Kirjallinen / suullinen ohjeistus vs Visuaalinen ohjeistus & tarina muotoiset tehtävänannot vs funktionaaliset vaatimukset.

- Mistä / keneltä vaatimukset tulevat tehtäville ? Mikä on kehittäjän oma rooli vaatimusten laatimisessa ?

-Jos tehtävät tulevat asiakkailta, onko kehittäjä yleensä itse yhteydessä asiakkaaseen, vai toimiiko joku välikätenä ?

Ohjelmistokehitysprosessi, toteutus päätökset / suunnittelu

Miten ohjelmiston suunnittelu tapahtuu ? Miten tehdään päätöksiä ohjelmiston rakenteesta ja valitaan käytettävät teknologiat ja ratkaisut ?

Mikä on haastateltavan oma rooli suunnittelussa ?

Mistä tai keneltä tarve ohjelmiston suunnittelulle yleensä kumpuaa ? Kehittäjältä itseltään, vai taustaorganisaatiosta / muilta kehittäjiltä?

Miten haastateltava tunnistaa kriittiset päätökset jotka vaativat suunnittelua tai katsomista isommalla porukalla (jos katsoo sellaisia olevan olemassa)?

Millaisia dokumentteja / artifakteja / whiteboard piirrustuksia suunnittelussa syntyy ja kuinka niitä käytetään ?

Kommunikaatio

Mitä kommunikaatio välineitä haastateltava käyttää, kun ohjelmistoa suunnitellaan yhdessä tai tehdään suunnittelupäätöksiä. (Esim face to face / virtuaali palaverit / flowdock / email)

Kuinka hyvin haastateltava kokee eri kommunikaatio välineiden toimivan ?

Ohjelmistokehitysprosessi, dokumentointi

Kuinka paljon haastateltava pyrkii dokumentoimaan tekemiänsä ratkaisuja ja toteutuksia ? Suosiiko haastateltava fyysisiä vai digitaalisia dokumentteja ?

Haasteet ja ongelmat

Onko Floweb Oyn kehitystyössä tällä hetkellä joitain ongelmia jotka hidastavat tai vaikeuttavat kehitystyötä.

Näkeekö haastateltava haasteita tulevaisuudessa

Arkkitehtuurin käyttötarkoituksia, onko ongelmia näiden kanssa (näitä voi kysellä jos haastateltava ei keksi mitään ongelmia).

Blue-Print (suunnittelukaavio), tietääkö kehittäjä mitä on tekemässä.

Roadmap - miltä yrityksen tulevaisuus näyttää ja mitä prioriteetit ovat

Kommunikaatiokanava - Näkymä siihen miltä järjestelmä näyttää

Työnjako, kuka tekee mitä ja tuleeko kaikki tehdyksi ?

Laadunvarmistus - osataanko järjestelmien toimivuus taata riittävän aikaisin ?

Riskien hallinta - ovatko kehitys ratkaisuihin liittyvät riskit tiedossa ?

Lopuksi

Haluaako haastateltava palata vielä johonkin aikaisempaan kohtaan ?

Onko haastateltavalla esimerkkejä ongelmista joita ei vielä ole noussut esiin ? Mitkä ovat haastateltavan mielestä kehitystyön isoimmat haasteet ?